Our Ref.: 723-1443

U.S. PATENT APPLICATION

Inventor(s): Patrick Link
Scott Elliot

Invention: HAND-HELD VIDEO GAME PLATFORM EMULATION

NIXON & VANDERHYE P.C.
ATTORNEYS AT LAW
1100 NORTH GLEBE ROAD
8TH FLOOR
ARLINGTON, VIRGINIA 22201-4714
(703) 816-4000
Facsimile (703) 816-4100

SPECIFICATION

TITLE

HAND-HELD VIDEO GAME PLATFORM EMULATION

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application is a divisional application of application serial no. 09/723,322 filed 11/28/2000 (Docket No. 723-950), now U.S. Patent No. _____. This application is related to copending commonly-assigned application serial no. 09/722,410 filed 11/28/00 entitled PORTABLE VIDEO GAME SYSTEM (Docket No. 723-951), which is a continuation-in-part of Application No. 09/627,440, filed 7/28/00. This application is also related to copending commonly-assigned application serial no. 09/321,201 of Okada et al filed 5/27/99 entitled "Portable Color Display Game Machine and Storage Medium for The Same". Priority is also claimed from provisional application number 60/233,622 filed 9/18/00 under attorney docket no. 723-932 entitled "Method and Apparatus for Emulating a Portable Game Machine." Each of these related applications is incorporated herein by reference.

FIELD

[0002] This invention relates to systems, methods, techniques, data structures, and other features for running software applications including but not limited to video games on platforms different from the ones the software is intended or designed to run on.

BACKGROUND AND SUMMARY

[0003] Nintendo's GAME BOY® hand-held video game platforms have been extraordinarily successful. Nintendo released the first GAME BOY® in the late 1980s. Since then, this product and its successors (GAME BOY COLOR ® and GAME BOY ADVANCE®) have captured the imaginations of millions of video game players throughout the world.

[0004] A wide number of different software applications (including but not limited to video games) have been designed to run on these platforms. People

throughout the world enjoy these applications every day. One can see them being used on subways, at sports arenas, after school, and in a number of other contexts. See Figure 1A.

[0005] Nintendo's GAME BOY®, GAME BOY COLOR® and GAME BOY ADVANCE® are examples of platforms having specialized hardware that is optimized for low cost, excellent performance and good graphics. These devices are not really general purpose computers; rather, they are special-purpose devices with specialized capabilities particularly adapted to video game play. These special capabilities provide low cost and exciting video game play action with good graphics and sound.

[0006] While GAME BOY® platforms are inexpensive and have long battery life, there may be situations in which it would be desirable to play or use applications developed for GAME BOY® on other platforms. For example, an airline, train or other vehicle passenger might want to play video games during a long journey. As shown in Figure 1B, airlines are installing seat-back computer displays into the backs of airline seats. Such seat-back displays may provide a low cost personal computer including a processor, random access memory, liquid crystal display and input device(s). Similar displays could be installed in other vehicles (e.g., trains, ships, vans, cars, etc.) or in other contexts (e.g., at walk-up kiosks, within hotel rooms, etc.). It would be desirable under certain circumstances to allow users to execute all sorts of different applications including GAME BOY® video games and other applications using the general-purpose computer capabilities of such seat-back or similar display devices.

[0007] Personal computers have also proliferated throughout the world and are now available at relatively low cost. A trend has shifted some entertainment from the home television set to the home personal computer, where children and adults can view interesting web pages and play downloaded video games and other applications. In some circumstances, it may be desirable to allow users to play GAME BOY® video games on their home personal computers (see Figure 1C).

[0008] A wide variety of so-called personal digital assistants (PDA's) have become available in recent years. Such devices now comprise an entire miniature computer within a package small enough to fit into your pocket. Mobile cellular telephones are also becoming increasingly computationally-intensive and have better displays so they can access the World Wide Web and perform a variety of downloaded applications. In some circumstances, it may be desirable to enable people to play GAME BOY® video games and other GAME BOY® applications on a personal digital assistant, cellular telephone or other such device (see Figure 1D).

[0009] The special-purpose sound and graphics circuitry provided by the GAME BOY® platforms is not generally found in the various other platforms shown in Figures 1B, 1C and 1D. Providing these missing capabilities is one of the challenges to running a GAME BOY® video game (or other GAME BOY® application) on these other target platforms.

[00010] Another challenge relates to instruction set compatibility. Nintendo's GAME BOY® is based on an older, relatively inexpensive microprocessor (the Zilog Z80) that is no longer being used in most modern general purpose computer systems such as personal computers, seat-back displays and personal digital assistants. The Z80 instruction set (the language in which all GAME BOY® games and other GAME BOY® applications are written in) is not directly understood by the more modern Intel microprocessors (e.g., the 8086, 80286, 80386, Pentium and other processors in the Intel family) that are now widely used and found in most personal computers, seat-back displays, personal digital assistants, and the like. While it is possible to "port" certain GAME BOY® games or other applications to different microprocessor families (e.g., by cross-compiling the source code to a different target microprocessor), there may be an advantage in certain contexts to being able to play or execute the same binary images stored in GAME BOY® cartridges on target platforms other than GAME BOY®.

[00011] One way to provide a cross-platform capability is to provide a GAME BOY® software emulator on the target platform. Generally, a software emulator is a

computer program that executes on a desired target platform (e.g., a seat-back display device, a personal computer or a personal digital assistant shown in Figures 1B-1D) and uses software to supply native platform capabilities that are missing from the target platform. For example, a software emulator may perform some or all of GAME BOY®'s specialized graphics functions in software, and may interface with whatever graphics resources are available on the target platform to display resulting images. A software emulator may translate or interpret Z80 instructions so the microprocessor of the target platform can perform the functions that GAME BOY® would perform if presented with the same instructions. The software emulator may include software code that emulates hardware capabilities within the GAME BOY® circuitry (e.g., audio and/or graphics processing) and/or translate associated GAME BOY® application requests into requests that can be handled by the hardware resources available on the target platform. For example, the target platform may include a graphics adapter and associated display that is incompatible with GAME BOY®'s graphics hardware but which can perform some of the basic graphics functions required to display GAME BOY® graphics on a display.

[00012] A number of GAME BOY® emulators have been written for a variety of different platforms ranging from personal digital assistants to personal computers. However, further improvements are possible and desirable.

[00013] One area of needed improvement relates to obtaining acceptable speed performance and high quality sound and graphics on a low-capability platform. A low-capability platform (e.g., a seat-back display or a personal digital assistant) may not have enough processing power to readily provide acceptable speed performance. Unless the software emulator is carefully designed and carefully optimized, it will not be able to maintain real time speed performance when running on a slower or less highly capable processor. Slow-downs in game performance are generally unacceptable if the average user can notice them since they immediately affect and degrade the fun and excitement of the game playing experience.

[00014] Performance problems are exacerbated by the penchant of some video game developers to squeeze the last bit of performance out of the GAME BOY® platform. Performance tricks and optimizations within a GAME BOY® application may place additional demands on any emulator running the application. Some prior art emulators provide acceptable results when running certain games but unacceptable results (or do not work at all) for other games. An ideal emulator provides acceptable results across a wide range of different games and other applications such that the emulator can run virtually any game or other application developed for the original platform.

[00015] Another challenge to designing a good software emulator relates to maintaining excellent image and sound quality. Ideally, the software emulator running on the target platform should be able to produce graphic displays that are at least the same quality as those that would be seen on the native platform. Additionally, the color rendition and other aspects of the image should be nearly if not exactly the same. Sounds (e.g., music and speech) from the emulator should have at least the same quality as would be heard on the original platform. All of these capabilities should be relatively closely matched even on platforms with radically different sound and graphics hardware capabilities.

[00016] One prior attempt to develop a video game platform emulator is disclosed in U.S. Patent No. 6,115,054 to Giles. That patent describes a general purpose computer based video game platform software emulator including an execution skipping feature that evaluates the ability of the general purpose computer to generate video frames fully synchronized with the target platform computer system. If the evaluation determines that the emulator is falling behind the target system, the emulator executes only a first subset of the graphics commands while skipping execution of a second subset of graphics commands so as to partially render the frame. For example, the patent discloses fully executing certain graphics commands while partially executing others (e.g., clipped drawing commands) to provide a partial rendering of the frame. One disadvantage to the approach described in the Giles

patent is that partial rendering of a frame can lead to uncertain imaging results that will degrade the quality of the image being produced by the emulator.

[00017] The present invention solves these and other problems by providing a unique software emulator capable of providing acceptable speed performance and good image and sound quality on even a low-capability target platform such as a seat back display for example.

[00018] The preferred embodiment software emulator provided by this invention maintains high-quality graphics and sound in real time across a wide variety of video games and other applications -- and nearly duplicates the graphics and sound that would be experienced by a user of the GAME BOY®, GAME BOY COLOR® and/or GAME BOY ADVANCE® platform running the same game or other application. The preferred embodiment emulator achieves this through a unique combination of features and optimizations including, for example:

- use of a virtual liquid crystal display controller (state machine) to maintain real time synchronization with events as they would occur on the native platform,
- use of a hardware-assisted bit BLIT memory transfer operation to efficiently transfer graphics information into video memory,
- pre-computed translation table for translating native platform graphics character formats into formats more compatible with standard graphics adapters,
- emulation of native platform color palette information to provide compatibility with games and other applications that change color palettes within a frame,

  emulation of major registers and other hardware-based memory structures within the native platform in RAM under software control,

use of a jump table able to efficiently parse incoming binary instruction formats,

- use of a unique page table to control memory access by remapping memory access instructions into different memory locations and/or function calls,
- availability of a ROM protection function to eliminate ROM overwriting during emulated operations,
- responsive to video game compatibility modes and registration data,
- models native platform using state machine defining search, transfer, horizontal blank and vertical blank states,
- cycle counter to determine when a modeled state has expired and transition to a new state is desired,
- selective frame display update skipping while maintaining execution of all instructions to maintain state information while minimizing game play slowdowns,
- optional NOP loop look ahead feature to avoid wasting processing time in NOP loops,
- redundant emulated RAM and ROM storage to optimize execution efficiency,
- separate page tables for read and write operations,
- modeling of native microprocessor registers as a union of byte, word and long register formats,

modeling native instruction CPU flags to allow efficient updating after operations are performed by target platform microprocessor,

mapping emulated program counter into target platform microprocessor general purpose register,

- reads and writes via index register go through pointer tables to increase execution efficiency,
- adaptable input controller emulator to provide user inputs from a variety of different user input devices,
- emulated object attribute memory, and
- use of screen memory buffers larger than screen size to increase paging efficiency by eliminating clipping calculations and using the hardware BitBlt to transfer a subset of the memory buffer to displayed video memory.

BRIEF DESCRIPTION OF THE DRAWINGS

[00019] These and other features and advantages provided by the invention will be better and more completely understood by referring to the following detailed description of presently preferred embodiments in conjunction with the drawings, of which:

[00020] Figure 1A shows someone playing a Nintendo GAME BOY® portable video game platform;

[00021] Figures 1B-1D show various different target platforms that could be used to emulate the Figure 1 GAME BOY®;

[00022] Figure 2 is a block diagram of an example software emulator architecture;

[00023] Figure 2A is a flow chart of example overall software emulator steps;

[00024] Figure 3 is a block diagram of example functional models of the Figure 2 emulator;

[00025] Figure 4 is a block diagram of example emulator memory objects/data structures;

[00026] Figure 5 shows an example emulated cartridge read only memory data structure;

[00027] Figure 6 shows example compatibility modes;

[00028] Figure 7 shows example registration data locations;

[00029] Figure 8 shows an example virtual liquid crystal display controller state machine state diagram;

[00030] Figures 9A-9B show example virtual LCD controller emulation/control registers;

[00031] Figure 9C shows example state machine cycle parameters;

[00032] Figure 10 shows an example flow diagram of an emulated liquid crystal display controller;

[00033] Figure 11 shows an example op code jump table;

[00034] Figure 12 shows example emulation of a particular (NOP) instruction;

[00035] Figure 13 shows an example page table;

[00036] Figure 14 shows an example memory access operation;

[00037] Figure 15 shows example read and write pointer tables;

[00038] Figure 16 shows example virtual microprocessor registers;

[00039] Figure 17 shows an example HL register write optimization;

[00040] Figure 18 shows an example input controller emulation register set;

[00041] Figures 19A and 19B show example additional emulator control registers;

[00042] Figure 20 shows an example graphics emulation optimization;

[00043] Figure 21 shows an example native character data;

[00044] Figure 22 shows example pre-rendered un-colorized "bit-map-ized" character tiles;

[00045] Figure 23 shows example graphics object pointers;

[00046] Figure 24 shows example emulated object attribute memory;

[00047] Figure 25 shows an example video memory transfer process;

[00048] Figure 26 shows example graphics mode selectors;

[00049] Figure 27 shows example screen layouts;

[00050] Figure 28 shows example VGA mode control parameters; and

[00051] Figure 29 shows example graphics engine register indices.

DETAILED DESCRIPTION

[00052] Figure 2 shows an example software emulator 100 provided by a preferred embodiment of the invention. Emulator 100 is designed to operate on a target platform of the type shown in Figure 1B above, but could run on any desired platform including, for example, the target platforms shown in Figures 1C and 1D.

[00053] In the example embodiment, the target platform includes:

- a microprocessor (e.g., an Intel 386);
- a disk or other file system 52;
- a keypad interface 54 coupled to a handheld controller 56;

a sound blaster or other audio interface card 58 coupled to a loud speaker or other sound transducer; and

- a VGA or other graphics adapter 62 that outputs video information to a display 64 such as a liquid crystal display screen or video monitor.

[00054] Emulator 100 (which executes on the target platform microprocessor and uses the resources of the target platform) receives the binary image of a game (or other application) file 66 stored on disk or other file system 52 (Figure 2A block 70). Emulator 100 parses and interprets this binary image (Figure 2A block 72). Emulator 100 also receives user inputs from handheld controller 56 via target platform keypad interface 54 (Figure 2A block 74). In response to these inputs, emulator 100 generates sound commands for the audio adapter 58 (Figure 2A block 76) and generates graphics commands for application to the video graphics adapter 62 (Figure 2A block 78) -- creating sounds on audio transducer 60 and images on display 64. These sounds and images nearly duplicate what one would hear and see if running file 66 on a native GAME BOY® platform.

[00055] In the example embodiment, the game file binary image 66 can be a video game or any other application that can run on a GAME BOY®, COLOR GAME BOY® or GAME BOY ADVANCE®. Binary image 66 includes binary audio commands and binary graphics commands, compatible with a GAME BOY® native platform but which are not compatible with the application programming interface features of audio interface 58 and VGA adapter 62. Emulator 100 interprets those graphics commands and sound commands, and generates a corresponding sequence of graphics and sound commands that are understandable by and compatible with the audio and sound capabilities of the target platform.

[00056] In the example embodiment, emulator 100 includes a virtual microprocessor core 102. Virtual microprocessor core 102 interprets instructions within the binary game file 66 that would be executed by the actual GAME BOY® native platform (Z80) microprocessor (Figure 2A block 72), and provides a corresponding sequence of microprocessor instructions for execution by the target

platform microprocessor (which in the general case, is different from the microprocessor found in GAME BOY® and does not understand and is incompatible with the native platform microprocessor instruction set).

[00057] Virtual microprocessor core 102 receives inputs from a keypad emulation block 104 (Figure 2A block 74). Block 104 in turn, receives interactive inputs from the user via target platform keypad interface 54. Keypad emulator block 104 emulates the GAME BOY® control input circuitry and associated functionality and translates inputs received from the target platform keypad interface -- which may have a different set of control inputs and configurations from that found in a GAME BOY® native platform.

[00058] Virtual microprocessor core 102 also communicates with sound emulation block 106 and graphics emulation block 108. Sound emulation block 106 emulates or simulates the sound generation circuitry within a GAME BOY®, GAME BOY COLOR® and/or GAME BOY ADVANCE® to provide a set of sound commands for application to the target platform sound adapter 58 (Figure 2A block 76). Graphics emulation block 108 emulates or simulates the hardware acceleration and other graphics circuitry found within a GAME BOY®, GAME BOY COLOR® and/or GAME BOY ADVANCE® platform to provide a set of graphics commands for application to a target platform graphics adapter 62 (Figure 2A block 78).

[00059] In the example embodiment, virtual microprocessor core 102 also includes a virtual liquid crystal display controller 103 used for the purpose of maintaining timing. Events within the GAME BOY®, GAME BOY COLOR ®, and GAME BOY ADVANCE ® native platforms are generally driven by activities relating to updating the liquid crystal display every one-sixtieth of a second. The example embodiment of emulator 100 emulates the native platform liquid crystal display controller (Figure 2A block 80) in order to synchronize events occurring within the emulator with emulated events that would occur within a GAME BOY®, GAME BOY COLOR ®, and/or GAME BOY ADVANCE ® native platform. As will be described below in detail, the virtual liquid crystal display controller 103 of the

example embodiment does not actually perform any display functions, but rather is used to tell emulator 100 what would be going on in terms of display timing on a real GAME BOY®, GAME BOY COLOR ®, or GAME BOY ADVANCE ®. A virtual liquid crystal display controller 103 allows emulator 100 to synchronize its pace with what the pace of a real GAME BOY®, GAME BOY COLOR ®, and/or GAME BOY ADVANCE ® native platform would be running the same application file 66. Virtual liquid crystal display controller 103 may be viewed as a software-implemented model of the event timing sequence of a GAME BOY®, GAME BOY COLOR ®, and/or GAME BOY ADVANCE ® native platform.

[00060] Emulator 100 also includes an emulated random access memory 110, an emulated read only memory 112, and an emulated memory bank controller (MBC) 114. Emulated random access memory 110 and emulated read only memory 112 provide memory storage locations within the (read/write) random access memory 68 of the target platform. The emulated random access memory 110 emulates or simulates the random access memory of a GAME BOY®, GAME BOY COLOR® and/or GAME BOY ADVANCE®, and the emulated read only memory 112 emulates or simulates the read only memory within the game cartridge of a GAME BOY®, GAME BOY COLOR® and/or GAME BOY ADVANCE® (Figure 2A block 82). The emulated memory bank controller 114 emulates or simulates the hardware memory bank controller (bank switching) circuitry found within certain a GAME BOY®, GAME BOY COLOR® and/or GAME BOY ADVANCE® game cartridges.

[00061] Figure 2A shows example steps performed by emulator 100. The emulator receives a binary game image (block 70) and activates any game-title-specific emulator options (block 71). The example emulator 100 parses and interprets the game binary image (block 72) and receives user inputs (block 74). The example emulator 100 generates sound commands (block 76) and graphics commands (block 78). The example emulator 100 emulates a native liquid crystal display controller (block 80) and native memory (block 82).

Example Emulator Functional Modules

[00062] Figure 3 shows a breakdown of example illustrative functional modules used to implement the Figure 2 emulator in software. These functional modules include:

- run game module 120,
- emulate module 122,
- draw_CGB module 124,
- draw_DMG module 126,
- draw_AGB module 128,
- ROM authentication check ("ROM REG") module 130,
- video module 132,
- VGA module 134,
- buttons module 136,
- sound module 138,
- no write module 140,
- port mode module 142,
- CGB RAM module 144,
- DMA module 146,
- MBC module 148,
- SIO module 150,

ADDPTRS module 152, and

timer module 154.

[00063] The example functional modules shown in Figure 3 provide various functions that can be called by name from other parts of the emulator code. Each of these functional modules may be implemented with a C or C++ and/or assembler function or other routine in one example implementation. In this particular implementation, the entire executable file (the aggregate of all modules) is designed as a DOS protected mode application that runs with a minimum number of drivers to maximize efficiency.

[00064] The run game functional module 120 loads the game file 66 into emulated ROM 112 and then calls the emulate functional module 122 (Figure 2A block 70). The run game module 120 may also by itself (or in conjunction with an additional function if desired) initialize each of the hardware-handler modules within the emulator 100. Emulate functional module 122 is the main emulation loop and is executed until the user quits the game or other application.

[00065] In the example embodiment, the draw functional modules 124, 126, 128 perform the task of drawing graphics objects generated by emulator 100 by sending graphics commands to the graphics adapter 62 (Figure 2A block 78). For example, the draw_CGB functional module 124 may draw each of 144 color background lines of the COLOR GAME BOY® on the screen and may also by itself (or in conjunction with another module) draw the moving objects after the background has been drawn. The draw_DMG functional module 126 performs a similar drawing task for original GAME BOY® games and other applications, and the draw_AGB functional module 128 performs similar drawing tasks for GAME BOY ADVANCE® games and other applications. Example emulator 100 is capable of emulating any/all of a number of different platforms across the Nintendo GAME BOY® product line.

[00066] In this example, the ROM check ("ROM REG") functional module 130 is used to check (and/or display) registration data within the game file 66. This functional module 130 is used to ensure, for example, proper authorization on the part of the user before game play is allowed. In another embodiment, the ROM registration

module does not do anything regarding user authorization, but just reads the ROM registration data in the game file, sets emulator variables and optionally displays the registration data on the screen. A game file validation function may be included in the ROM registration module to validate the game file, not the user.

[00067] The video functional module 132 is used in the example to transfer character graphics data. The functions in the video module 132 perform character bitmap translation for any type of write to the character RAM area, whether it is a direct write from the CPU or a DMA transfer. Functions in the video module also handle the RAM bank switching register for character data areas, control and status registers for the LCD controller and palette registers for both CGB and DMG modes. When a game file 66 instruction calls for a direct memory access data transfer of character information into the GAME BOY® character RAM space, video functional module 132 performs a character bit map translation into a portion of emulated RAM 110 to prepare graphics characters for display. The video functional module 132 may, by itself or in conjunction with another functional module, place appropriate function pointers into appropriate input/output read/write tables for all of the register handling functions that should be performed.

[00068] In the example embodiment, the VGA functional module 134 is used to set the appropriate video mode of the target platform graphics adapter 62. In addition, this VGA functional module 134 may be responsible for transferring full screens of graphics data to VGA graphics adapter 62 under certain circumstances (e.g., if a hardware-assisted bit BLIT operation is not available on the target platform).

[00069] The buttons functional module 136 is responsible for getting the keypad data from keypad interface 54 and writing this data into a set of input interface registers that emulate actual hardware interface registers within GAME BOY®, GAME BOY COLOR® and/or GAME BOY ADVANCE® .

[00070] The sound functional module 138 in the example embodiment generates and writes appropriate sound information to the target platform sound adapter 58 by translating writes to the virtual sound registers to appropriate sound

information for the target platform sound adapter (Figure 2A block 76). The sound functional module 138, by itself or in conjunction with another functional module, may also be used to put function pointers into appropriate input/output, read/write tables for all of the register handling functions performed by the sound functional module.

[00071] In this example module, the no write functional module 140 protects the emulated ROM 112 from being written to (thus making sure this memory segment is emulated as a read only memory as opposed to a read-write memory during game play). The no write functional module 140, by itself or in conjunction with an additional functional module, may place appropriate function pointers into the appropriate input/output read/write tables for all of the register handling functions in the no write functional module.

[00072] The port mode functional module 142 emulates a CPU timer and provides a keypad handler. It has functions that handle the keypad, the timers, and the CPU speed control (e.g., to provide a CPU speed change operation since COLOR GAME BOY® operates twice as fast as GAME BOY® and GAME BOY ADVANCE operates still faster). The port mode functional module 142 may also set appropriate function pointers or call an additional function module(s) to perform this task. The main function of the CPU timer is to generate CPU interrupts at specified intervals. Registers to specify this interval are handled in the port/mode module. There are a couple of registers that provide real-time views of a free-running counter. These registers can be emulated by returning a random number. This is only a partial emulation (a random number is not a real time value). However, the most common use of these registers by games is to generate a random number by looking at a fast clock at an arbitrary point in time. It is therefore possible to completely satisfy such games by providing a random number as opposed to a real time clock indication. A more accurate emulation can be provided if a game requires the real-time view of the counter actually provided in the native hardware.

[00073] The CGB RAM functional module 144 emulates the COLOR GAME BOY® RAM to provide (additional) emulated RAM 110. DMA functional module 146 performs direct memory access transfers between the various emulated storage resources within emulator 100 -- thereby emulating the GAME BOY® native platform DMA controller. The MBC functional module 148 emulates the native platform memory bank controller to provide emulated MBC 114.

[00074] The SIO functional module 150 emulates a serial input/output port available on a GAME BOY®, GAME BOY COLOR® and/or GAME BOY ADVANCE® platform (e.g., to provide a "game link" operation whereby plural platforms can exchange data over a cable or other communications interface). The ADDPTRS functional module 152 performs the task of registering various handlers for operation (in particular, it may contain a single function that all hardware support modules call to register their memory/function pointers in an I/O handler table, and accomplishes this by registering pointers for reading and writing to I/O addresses). The timer functional module 154 implements the virtual liquid crystal display controller 103 by maintaining an emulated state machine that keeps track of the state and associated timing information of a GAME BOY®, GAME BOY COLOR® and/or GAME BOY ADVANCE® platform. Timer module 154 thus allows the target platform (which may operate at a completely different speed from the original platform) to maintain a sense of the event timing as those events would occur on the native platform --ensuring that emulator 100 provides event timings that are consistent with the native platform. Without such timing information, the speed of the application's graphics and/or sound might be different on the emulator 100 as compared to on the original platform -- resulting in an unsatisfying game play experience.

Example Memory Objects And Data Structures

[00075] Figure 4 is a block diagram of exemplary memory objects/data structures that emulator 100 maintains in the random access memory 68 of the target platform. In some cases, these data structures emulate hardware resources of the

native platform. In other cases, these data structures do not correspond directly to any part of the native platform but instead provide support for optimized execution of emulator 100.

[00076] Figure 4 shows the following exemplary data structures:

- emulated "read only memory" 112,
- emulated random access memory 110,
- register table 160,
- raw character data buffer 162,
- translator 163
- "bit-map-ized" character data buffer 164,
- background data buffer 166,
- off screen display buffer 168,
- on screen display buffer 170,
- memory bank switched (cartridge) RAM buffer 172,
- object attribute memory buffer 173,
- object index data structure 174,
- CGB RAM buffer 175
- object enable data structure 176,
- page table 178,

jump table 182,

various color palettes including a high priority background palette 184a, a low priority background palette 184b, and an object color palette 184c for emulating COLOR GAME BOY®, and

- various monochrome color (gray scale) palettes for GAME BOY® monochrome game emulation, including a background palette 186a, an object 0 palette 186b and an object 1 palette 186c.

[00077] The Figure 4 data structures may be globally-defined memory arrays.

[00078] The main RAM array 110 is, in one example, a generic 64K memory array used for any non-paged address space. A CGB buffer 175 is used to emulate the internal RAM banks for COLOR GAME BOY®. MBC RAM 172 is used to emulate the random access memory that may be provided within certain game cartridges.

[00079] The object index array 174 may be used for sorting moving objects.

[00080] The object enable array 176 may include a flag for each display line indicating that drawing of moving objects was enabled for that line (flags may be sent/queried as the background is drawn).

[00081] Page table 178 may comprise a 64K table of pointers to the base pointers that handle each address, and may be used to reestablish the program counter on jumps, calls, returns, etc.

[00082] Page table 178 may be used for making pointer adjustments to both the program counter and the stack pointer. In another embodiment, a separate stack table comprising for example a 64K table can be used in a similar manner to page table 178, but with a coverage of each base pointer extending one address higher and used to reestablish the base of the stack pointer when it is manually changed.

[00083] The ROM pages 112 may be used to emulate the cartridge read only memory arrays (in the example embodiment, this ROM array is twice as big as the actual ROM pages since the bottom half is always duplicated).

[00084] The raw character data array 162 is used to store raw character data, and the further character data array 164 is used to store corresponding "bit map-ized" character data. A translator 163 is used to provide precomputed translation data for translating the raw character data 162 into the bit mapped character data 164. Different sets of pointers are used for each page and addressing mode in this example. The background data buffer 166 is used to store background data in pages 0 and 1.

[00085] The off screen buffer 162 (which may comprise an entry of 192 x 160 x 2) may be used to compose images off screen. This buffer may not be needed when a bit BLIT capability is available within the hardware of the target platform.

[00086] Color background palettes 184a, 184b comprise two sets of eight palettes, one for high priority background pixels and the other for low priority background pixels. Color object palette 184c provides object palette data to emulate the COLOR GAME BOY® object color palette (one set of eight palettes may be provided). GAME BOY® color palettes 186a, 186b, 186c emulate the monochrome GAME BOY® palettes, with background palette 186a providing four background palette data entries and object palettes 186b, 186c comprising object palette data for object 0 and object 1 (four entries per palette). The native COLOR GAME BOY® platform has selectable palettes for "colorizing" monochrome GAME BOY® games -- and this capability may also be emulated by, for example, changing the color entries within palettes 186a, 186b, 186c. In another embodiment, these palettes 186 may be preassigned to provide certain default colors (e.g., red objects on a green background).

[00087] Jump table 182 is used to facilitate the parsing and execution of target instructions by emulator 100, as is explained below.

Example Emulated Cartridge ROM 112

[00088] Figure 5 shows an example emulated cartridge ROM 112. In the native platform, the cartridge ROM may have a number of banks up to a maximum. Preferred embodiment emulator 100 emulates each of these banks with a different RAM page 112(1), 112(2), 112(n). In one example embodiment, the number of pages

that may be allocated can be fixed (e.g., to a maximum of n=256) to provide static allocation for a four-megabyte game. In another embodiment, the number of ROM pages to allocate can be determined dynamically based on the particular game or other application.

[00089] In the example embodiment, the lower 16K in each allocated ROM page 112(1), ... 112(n) is duplicated to facilitate page selection and reduce page swapping. A ROM page selection pointer 202 is used to select the current ROM page, and a ROM page count register 204 specifies the number of ROM pages loaded for the current game or other application. As mentioned above, the "no write" functional module 140 is used to protect the ROM space so that inadvertent write instructions within the application and/or emulator 100 do not succeed in overwriting emulated read only memory 112.

[00090] As mentioned above, the run game routine 120 is responsible for loading the game (application) file 166 into emulated ROM 112. Part of this loading operation loads particular compatibility information (see Figure 6) and registration data (see Figure 7) into the emulated ROM 112. The Figure 6 compatibility information is used to specify whether an application is compatible or incompatible with certain native platforms (e.g., compatibility with the COLOR GAME BOY® mode of emulator 100, or whether it can run exclusively on the COLOR GAME BOY® mode). This compatibility information is present in a normal binary game file 166 to provide instructions to the COLOR GAME BOY® platform; emulator 100 reads and takes advantage of this information in determining its own emulation mode. The registration data shown in Figure 7 is used in the example embodiment to ensure that game file 66 is authorized and authentic, and emulator 100 performs checks similar to those performed by the GAME BOY®, COLOR GAME BOY® and GAME BOY ADVANCE® native platforms (as well as possibly other security checks such as digital signatures, decryption, digital certificates, etc.) to ensure the user has proper authorization.

Example Virtual Liquid Crystal Controller 103 Implementation

[00091] In the example embodiment, emulator 100 uses an internal state machine to keep track of and emulate the states of an actual GAME BOY®, COLOR GAME BOY® or GAME BOY ADVANCE® platform during emulation operation. The emulator 100 could execute the instructions within game file 66 without keeping track of corresponding events within the native platform, but this would lead to loss of real time synchronization. In video game play, the pacing of the audio and video presentation is very important to the game play experience. Playing a game too fast or too slow will tend to destroy the fun of the game It is therefore desirable to emulate a game playing experience that is close to or nearly the same as the game playing experience one would have when running the application on the original native platform.

[00092] Emulator 100 accomplishes this result by maintaining liquid crystal display controller 103 providing a sequential state machine that is synchronized with event states that would occur on the original native platform. Emulator 100 synchronizes its operation to the state transitions within this internal state machine to maintain real time synchronization of game play.

[00093] Figure 8 shows an example four-state virtual state machine state transition diagram that can be maintained by virtual LCD controller 103. These states include:

- an object attribute memory search state 250,
- a memory transfer state 252,
- a horizontal blanking state 254, and
- a vertical blanking state 256.

Additional states (e.g., enable and disable) can also be provided.

[00094] In the example embodiment, the sequential progression through all four states 250-256 comprises a frame that results in the display of a new image on display 64. In the native platform, one frame comprises a vertical blanking state 256 and various repetitions of the hblank, OAM search and OAM transfer states 254, 250, 252 dependent on the number of lines (e.g., 144) within a frame. Because the native platform hardware is driven by line scanning operation of a liquid crystal display, so too is preferred embodiment emulator 100 driven by an emulated state machine that models the same line scanning and other time intervals to ensure proper game timing as the developers of the game intended it and as a user would see and experience a game on the native platform.

[00095] Within each line there is an hblank interval and associated state 254, as well as an OAM search state 250 (during which a native platform would search its object attribute memory for objects to be displayed on the next line) and an OAM transfer state 252 (during which a native platform transfers object character information into a line buffer for display). The table of Figure 9C shows example cycle parameters for the Figure 8 virtual state machine.

[00096] The preferred embodiment emulator 100 emulates a virtual state machine by maintaining the various registers shown in Figures 9A and 9B. The registers shown in Figure 9A generally comprise various registers used to keep track of the virtual state and operation of a liquid crystal display controller that is being emulated. In this example, emulator 100 emulates a liquid crystal display controller using the following registers:

- LCD cycle counter 260 (maintains the number of CPU cycles remaining before a transition to the next liquid crystal display controller phase/state should occur),

- liquid crystal display mode register 262 (maintains the current phase/state of the liquid crystal display controller including the various states shown in Figure 8 as well as an additional disabled and re-enabled state),

a liquid crystal display background enabled flag 264 (indicates whether the background should be drawn),

- a liquid crystal display window enabled flag 266 (indicates whether the current display window is enabled),
- a liquid crystal display object enabled flag 268 (indicates that the drawing of moving objects is enabled),
- a liquid crystal display big object flag 270 (indicates that objects are sixteen lines high instead of eight),
- a last object draw line register 272 (indicates the last line at which a direct memory access to object attribute memory occurred).

[00097] The Figure 9B timing registers are used to maintain the various parameters pertaining to the timing parameters associated with the Figure 8 virtual state machine. These registers include:

- a cycleshblank register 274 (specifying the number of virtual CPU cycles needed in the horizontal blanking period),
- a cyclesvblank register 276 (indicating the number of virtual CPU cycles needed in the vertical blanking period),
- a cycles OAM (search) register 278 (indicating the number of virtual CPU cycles needed in the OAM search period),
- a cycles transfer register 280 (indicating the number of virtual CPU cycles need in the liquid crystal display data transfer period),
- a cycles frame register 282 (indicating the number of virtual CPU cycles needed for an entire frame),
- a timer ticks register 284 (this comprises a master game timer and is incremented by interrupt every $1/60^{th}$ of a second),

a cycle counter 286 (this may be implemented by a local variable within the main emulation functional module 122 and is used to keep track of the current number of cycles within the frame),

- a timer target liquid crystal display counter flag 288 (this is a flag indicating when the cycle counter 286 reaches 0 in order to control the virtual liquid crystal display controller to transition to the next phase shown in Figure 8),
- a fast CPU flag 290 (a flag indicating that the emulated COLOR GAME BOY® CPU is running in double-speed mode),
- a do frame flag 292 (a flag indicating whether emulator 100 should draw the current and/or next video frame or skip drawing it),
- a timer cycle counter 294 (indicates the number of CPU cycles remaining before a timer interrupt should be asserted),
- a timer threshold register 296 (indicates the number of CPU cycles corresponding to the current timer interrupt period),
- a timer enable register 298 (a flag indicating that timer interrupts are enabled).

[00098] Figure 10 is a flow diagram of an example emulated liquid crystal display controller 103. This flow diagram uses the various registers shown in Figures 9A and 9B to implement the Figure 8 state machine. The Figure 10 flow diagram has been simplified for purposes of illustration; additional operations may occur in an actual implementation. As shown in Figure 10, the virtual state machine is initialized with an initial state by updating an express or implied state counter (state may be explicitly stored in register 262 or it may be implied through inline code for efficiency purposes if desired) (block 302). Then, the cycle counter register 286 is loaded with an appropriate number of cycles from the one of registers 274, 276, 278, 280 corresponding to the current state of the state machine (block 304, see Figures 8 and

9C). The cycle counter 286 is continually decremented at the emulated CPU rate (block 306) (as determined, for example, by the fast CPU flag 290) in response to timer ticks 284. This cycle counter 286 is continually compared with zero (decision block 308) to determine whether the current state is over. When the cycle counter has been decremented to zero (the "=" exit to decision block 308), the emulated LCD controller 103 transitions to the next state of the virtual state machine (see Figure 8) (block 310). In the example embodiment, the cycle counter register is decremented by a fixed amount for each CPU instruction emulated. The effect of double-speed CPU operation is accomplished by loading the cycle counter with twice the number for each LCD controller phase than would be loaded for single-speed operation. So the cycle counter gets decremented at the same rate (which is determined by the speed of the host CPU), but the CPU can run through twice as many cycles per LCD phase in double-speed mode. Since the game speed is governed by throwing in an appropriate wait time once per frame in the example embodiment, the game speed is correct for both fast and slow modes, but in fast mode the CPU can do twice as much work.

[00099] If the next state is vertical blanking ("yes" exit to decision block 312), then emulator 100 determines whether it is running behind (e.g., by determining the amount of time until the next timer interrupt is going to occur). Preferred embodiment emulator 100 tries to maintain the sixty frames-per-second screen update rate of the native platform. However, in one particular embodiment, it is not always possible (e.g., depending upon the particular game of other application being executed) to maintain a sixty frame-per-second rate on a slow target platform. In that example embodiment, emulator 100 dynamically scales back to a slower, thirty frame-per-second rate by setting the do-frame flag 292 ("yes" exit to decision block 314, block 316) which will have the result of entirely skipping the drawing of the next frame. In that example embodiment, this frame-skipping operation does not skip execution of any instruction from game file 66. All such instructions are executed by virtual microprocessor core 102 in order to continually maintain and update appropriate state information. Furthermore, this frame-skipping operation does not have the result, in the embodiment, of partially rendering the frame being skipped. For example, there is

no selective execution of certain graphics commands in a command buffer depending on whether or not the emulator is falling behind. In that example embodiment of emulator 100, the only operations that are skipped are internal emulator 100 operations of transferring graphic information to the VGA graphics adapter 62 and updating the display 64 -- resulting in the frame either being rendered or not being rendered. Since the GAME BOY platforms operate to render an entire new frame each $1/60^{th}$ of a second "from scratch", there is no need to partially render a frame for use in generating a next frame, and such a partial rendering would tend only to degrade speed performance and generate uncertain image results. A maximum of every other frame may be skipped in the example embodiment since using a frame update rate of less than $1/30^{th}$ of a second would noticeably degrade image quality.

[000100] In a further embodiment, the "dynamic-scaling" feature is omitted from the emulator 100 to allow better emulation of transparency-based images. It turns out there are some games that achieve transparency effects by enabling and disabling the visibility of entities on the screen at a 30fps rate (on for one frame, off the next). Allowing the emulator to skip "as needed" between 30fps and 60fps causes undesirable flickering in such games. In this alternate configuration, the emulator 100 may draw frames at either a fixed 30fps (skip drawing of every other frame) or a fixed 45fps (skip drawing of every third frame). Running at 30fps causes the object to either always be visible or never be visible, depending on which phase you hit on. This is less than perfect emulation, but actually is the best solution for at least some games. For example, the 45fps rate is currently used in certain games to make characters blink when they are hit by an enemy. Running at 45fps (which provides acceptable game speed in certain games but not many other CGB games) allows you to alternate between visible and invisible and provides a good flickering character. If the emulator could draw at 60fps, none of these problems would exist, but slow target hardware does not permit this. Luckily, 30fps provides good game play for most games. It is possible to modify a few bytes (the "game code", which the emulator does not use) in the ROM registration area of the game file to tell the emulator what frame

rate to use. There may be other game-specific emulation parameters put into the game file in the future.

Example Instruction Parsing/Execution By Virtual Microprocessor Core 102

[000101] In the example embodiment, the virtual microprocessor core 102 interprets the binary instruction formats of game file 66 (Figure 2A block 72). As mentioned above, the game file 66 binary instruction formats in the example embodiment are compiled for execution by a Z80 microprocessor of the native platform -- whereas the target platform on which emulator 100 runs may be any microprocessor (e.g., an Intel 8086 family microprocessor). In the example embodiment, the virtual microprocessor core 102 may include a binary instruction format parser implemented as a jump table (e.g., C or C++ "case" statement) that parses the binary op code portion of the incoming instruction and jumps to appropriate code that performs one or a series of steps that will cause emulator 100 to emulate the operation of that instruction. Figure 11 shows an example jump table flow based on the jump table 182 (which may be implemented as inline code if desired).

[000102] Those skilled in the art will understand that different native instructions can be emulated in different ways depending upon the particular instruction. Figure 12 shows an example flow diagram for emulation of an example "no operation" (NOP) instruction. In this Figure 12 example, an op code of "00" parsed by the Figure 11 process results in transferring control to the Figure 12 process for emulating the "no operation" instruction. On the native platform, a "no operation" instruction results in nothing happening (wait) for a CPU cycle. Within emulator 100, in contrast, certain tasks are performed in response to such a "no operation" instruction. For example, an emulated program counter (which is different from the target platform program counter and is used to emulate the program counter of the native platform) is incremented (block 322), and the cycle counter is decremented (see block 306, Figure 10). As shown in Figure 10, if the cycle counter is not greater than zero, a "timer" function is called to perform the steps of blocks 310-316 shown in

Figure 10. If the cycle counter is still greater than zero, then control returns to the Figure 11 operation to parse the next op code (block 324).

[000103] Some games and other applications make extensive use of "no operation" loops to maintain game timing. Somewhat surprisingly, such "no operation" loops can cause emulator 100 to run very slowly. To avoid this particular issue, it is possible for emulator 100 to include a dynamic code analyzer that "looks ahead" to the next few instructions surrounding a "no op" instruction to determine whether the game file 66 includes a "no op" loop. If emulator 100 determines that such a loop is present, then the emulator may intelligently use events other than a wait loop (e.g., setting a timer and waiting for it to expire, or relying on the virtual liquid crystal display controller 103) as alternate means for providing the requisite "wait loop" timing. This optimization can result in increased efficiency by preventing the emulator 100 from becoming bogged down with "no operation" wait loops. In other embodiments, no NOP-reduction analysis is implemented, and the only such technique implemented is to detect whether a loop was waiting for a transition of the LCD machine and automatically force the transition. The problem is that such a technique may work for some games, but could cause some games to malfunction.

Example Memory Access Instruction Emulation

[000104] Figure 13 shows an example page table 178 within the context of a memory map that also includes emulated RAM 110, 172, 175 and emulated ROM 112. This page table 178 is used in the example embodiment to process memory access commands within game file 66. In this example embodiment, some memory access (read or write) commands can be executed by performing the requested read or write operation on a specified location within memory. In such cases, page table 178 includes a memory pointer specifying a corresponding memory location -- remapping various read/write locations into other locations as defined within the emulator 100 (see Figure 14, blocks 332, 334, 336). In some cases, a read or write to a particular memory location will trigger the performance of a sequence of steps by emulator 100. As an example, a read by the game file 66 of a game controller input register of the

native platform may cause emulator 100 to execute a "key" function in order to poll the keypad interface 54 and get a user controller input value. The preferred embodiment page table 178 handles this situation by providing a zero-valued memory pointer within page table 178 (Figure 14, block 334) that causes the emulator to reference an associated "key" function pointer -- resulting in the calling of a "key" function (Figure 14, block 338). In this way, page table 178 efficiently maps native instruction memory accesses to the same or different memory locations within emulated memory and/or to calling a function that emulates a result which would occur on the native platform in response to such a memory access command.

[000105] Also as shown in Figure 13 and alluded to above, the emulated random access memory 172, 175, 110 and the emulated read only memory 112 may include multiple copies of the same information within the target platform random access memory 68 in order to provide more efficient paging and corresponding reduction in processing time.

[000106] Figure 15 shows implementation detail for one detailed implementation of page table 178. In this example, the page table may comprise two different tables 178a, 178b -- one for read memory accesses and one for write memory accesses. Each of these tables may be 64 kilobytes (or other convenient size). All memory accesses by virtual microprocessor core 102 are performed via these tables 178a, 178b. The code that is reading or writing first looks to see if there is a non-null value in the "PTR" element for the desired address. The "PTR" element is a pointer to the pointer that defines the base of the target platform memory array that applies for the desired address. If there is a non-null "PTR" value, de-referencing "PTR" and adding the desired address will get emulator 100 to the target platform address to read/write. If, on the other hand, the "PTR" value is null, that means that there is a handler function defined for reading/writing to the desired address. The handler function can be called via the "FUNCT" element of the appropriate table.

[000107] Different functions can be called for reading from and writing to the same address in this example arrangement, and different pointers may be used reading

from and writing to the same address. Similarly, a read operation with respect to a particular native address may cause a read from an active "PTR" memory mapped value whereas a write operation to the same address can invoke a handler function -- or vice versa. The flexibility provided by this arrangement simplifies the architecture of emulator 100 while providing an efficient way to execute instructions from game file 66.

Emulated Microprocessor Registers

[000108] Figure 16 shows example emulated registers within the virtual microprocessor core 102. In this example, the native (e.g., Z80) microprocessor registers are emulated with random access memory values within the target platform RAM 68 and/or actual registers internal to the target platform CPU. For example, it may be desirable to map certain emulated native microprocessor registers to target microprocessor registers for efficiency purposes (e.g., to map a program counter 350 to a general purpose register within the target platform CPU).

[000109] In the example embodiment, the program counter or program pointer 350 may include a current base pointer for the program counter as well as an offset portion. Similarly, a stack pointer 352 may include a base pointer for an emulated stack pointer to which may be added an offset (e.g., in a target platform register). Virtual microprocessor core 102 may further include a set of emulated native platform flags 354 including:

- a carry flag 354a,
- a half-carry flag 354b,
- an add sub flag 354c,
- a zero flag 354d.

[000110] In the example embodiment, emulated flags 354 are not in the same bit positions as the native platform flags, but rather they are in positions used by the

target platform processor. This allows emulator 100 to pass "virtual" flags to the target platform processor before performing operations that effect the flags. The target platform flags are retrieved into the virtual flag data structure 354 after the operation is performed.

[000111] In the example embodiment, the various native platform general purpose registers are defined in three separate data structures as bytes (block 356), words (block 358) and long words (block 360). The three structures 356, 358, 360 are bundled into a union so that emulator 100 can access a particular register as a byte, a word or a long word as needed. In the example embodiment, the program pointer 350 is not included because it is maintained as a C character pointer for maximum efficiency. The program counter or pointer can be declared as a local variable in the main emulation function 122, and the compiler preferably implements the program pointer 350 as a register in the target platform CPU as described above.

[000112] Some additional optimizations are possible when accessing the emulated registers shown in Figure 16. For example, the HL register within the native platform CPU is often used as an index register. As Figure 17 shows, it is possible for virtual microprocessor core 102 to "look ahead" by determining whether the indexed address is for a special hardware location in response to a write to the HL register (decision block 370) -- and to access page table 178 immediately in response to such an indexed address so that the corresponding memory pointer and/or function are available when a further instruction comes along that uses the HL register contents for an indexed operation (block 372). This optimization can save processing time. Indirect accesses via HL or any other 16-bit register (BC or DE) are all handled by referring to the I/O read/write handler tables in the example embodiment. One "look-ahead" technique the preferred embodiment emulator uses is the "prefetch queue" implemented by always fetching four bytes into a 32-bit target platform register each time. The low-order byte is the opcode the emulator is after, but many opcodes require one or two subsequent bytes as data or extended opcode. By having four bytes in a register, any opcode handlers that need subsequent bytes already have them in a CPU register.

[000113] Referring once again to Figure 16, the virtual microprocessor core 102 further includes a set of interrupt vectors and an interrupt master enable flag that are used to emulate the interrupt structure within a GAME BOY®, GAME BOY COLOR ® and/or GAME BOY ADVANCE® native platform. This interrupt vector (when enabled by the interrupt master enable flag) can be read to determine what portion of emulator 100 caused a particular interrupt (e.g., vblank, the liquid crystal display controller, a timer, button depression, or serial input/output). Emulator 100 provides an emulated interrupt controller that emulates the actual native platform interrupt structure in controller to maintain compatibility and event-driven functionality of game file 66.

Example Keypad Emulation

[000114] As shown in Figure 18, preferred embodiment emulator 100 provides keypad emulation 104 in the example embodiment through the use of certain data registers/flags including:

- a buttons direction register 380 that maintains the data for emulated direction keys,
- a buttons buttons register 382 that maintains the data for the emulated control buttons, and
- a buttons changed flag 384 that indicates that the button data has been changed.

[000115] In one example embodiment, the buttons direction register 380 and the buttons buttons register 382 encode various button parameters in certain bit positions as shown in Figure 18. As mentioned above, the buttons functional module 136 shown in Figure 3 may be used to retrieve inputs from keypad interface 54 and load them into the Figure 18 data structures for reading by virtual microprocessor core 102. These data structures and associated functionality emulate the hardware control input controller of the native platform by duplicating the register interface of the native

platform in software. Target platform controller device 56 may be any of a variety of different configurations including, for example, an SNES handheld controller, a keypad, or any other input device capable of interacting with a user. A "parallel port" register or indicator 388 may be used to define the type of keypad interface 54 (e.g., SNES controller adapter or keyboard) that will be used for the controller input on the target platform.

Miscellaneous Additional Virtual Microprocessor Data Structures/Functions

[000116] Figures 19A and 19B show example additional virtual microprocessor data structures. These data structures are used to provide a variety of different additional functionality in the example embodiment of emulator 100.

[000117] As shown in Figure 19A, preferred embodiment emulator 100 may include one or more game-specific emulation options that go into effect for particular games or other applications (Figure 2A block 71). As one example, an "options" data structure 402 may specify particular functions and/or features that could be activated selectively depending upon the particular application or game being supplied by game file 66. Such game-specific emulation options can improve efficiency by tailoring the operation of emulator 100 for particular applications or games on a dynamic, as-needed basis. While in some embodiments it would be best to avoid using game-specific options, in other examples it might be desirable to use such game-specific options to increase efficiency and/or functionality.

[000118] As shown in Figure 19A, one game-specific option might be using a single CGB_RAM memory pointer. Another game-specific option is the 30/45fps frame rate option described previously. Other game-specific options are possible.

[000119] Figure 19A also shows a "DMG only" flag 404 that is used in the example embodiment to indicate that the loaded game file 66 is COLOR GAME BOY® incompatible. This DMG only flag 404 (which is set or unset depending on the compatibility modes shown in Figure 6) is used to determine whether COLOR GAME BOY® functionality of emulator 100 is enabled or disabled. It is also possible

to provide a flag indicating that the stack pointer is allocated to a particular region of memory (e.g., fixed emulated COLOR GAME BOY® RAM). The flag that indicates that the stack pointer is pointing to a particular region of memory (fixed CGB RAM) is not a game-specific option in one example embodiment, and is set dynamically by the emulator 100.

[000120] A rumble pack flag 406 is used in the example embodiment to indicate whether the loaded game file 66 supports the rumble pack feature of certain native platform games.

[000121] The TSR interrupt register 408 in the example embodiment specifies the number of the DOS interrupt used for host-to-emulator communication.

[000122] A DMA source register 410 specifies a source address for emulated direct memory access operations, and a DMA destination register 412 specifies a destination address for emulated direct memory access operations. A memory base pointer 414 specifies a base pointer for non-paged memory 110.

[000123] Referring to Figure 19B, a register file including for example, various native platform registers emulated in software (RAM locations) is shown. Such registers include, for example, sound control registers ("NR10-NR52), a liquid crystal display controller register having the bit assignments shown, and a status register STAT having the bit assignments shown.

[000124] In terms of sound emulation, certain information written to the sound control registers may be straight-forwardly translated and passed on to the target platform sound adapter 58 using the particular API used by that sound adapter. Other sound generation commands are peculiar to the GAME BOY®, GAME BOY COLOR® and GAME BOY ADVANCE® native platforms, and need to be emulated using sound-producing functions. These sound-producing functions take advantage, as much as possible, of the target platform sound generation capabilities, but typically need to provide additional state information (e.g., implementation of a sound-generation state machine) in order to ensure sound timing synchronization.

Maintaining real time sound timing sound synchronization is especially important with voices -- which will sound unnatural if played back too fast or too slow. Unfortunately, voice reproduction may be difficult to achieve since the strict CPU timing necessary to play back voice takes up too much time in itself, and may not be possible to perform on a low-resource target platform. Games that rely on voice playback for a satisfactory game play experience may have to be excluded. In the example embodiment, the sound module translates writes to the virtual sound registers to appropriate sound information for the target platform sound adapter. If the sound library used does not provide for automatic termination of sounds after specified durations, then the emulator 100 may also be provided with the capability to terminate sounds at appropriate times.

Graphics Emulation

[000125] As described above, a graphics emulation 108 portion of emulator 100 in the example embodiment receives commands from the virtual microprocessor core 102 and performs responsive graphics tasks. This graphics emulation functionality performed by block 108 supplies capabilities normally supplied by the graphics acceleration hardware of the native platform.

[000126] One way to provide such graphics emulation 108 would be to nearly exactly implement, in software, each of the hardware structures of the native platform's graphics circuitry. This is not necessarily the best approach, however, since it may be more efficient to perform certain graphics-related tasks differently in software. Figure 20 shows an example of how the efficiency of preferred embodiment emulator 100 is enhanced by handling character data differently than the way it is handled in the native platform.

[000127] In the Figure 20 example, a pre-computed translation table 163 is used to translate "raw" character data within an array 162 into a "bit mapped-ized" character data format for storage into buffer 164. Figures 21 and 22 further illustrate this feature. The Figure 21 representation shows a portion of the "raw" character data

buffer 162 storing the character data bit planes as they are typically maintained by the native platform. Pre-computed translator 163 translates this raw character data representation into a differently-ordered and organized, bit mapped character data representation more like the format found in a conventional bit map (.bmp) file. This character data reorganization is useful in minimizing processing time required to output character graphics data to the video adapter 62. The Figure 22 "bit mapped-ized" representation is more compatible with VGA and other commonly-used video adapter hardware, and the pre-computation of translator 163 allows this data reorganization to occur in a straight forward manner in advance of the time when the graphics data is sent to the graphics adapter 62.

[000128] Figure 23 shows a number of example graphics object pointers used by graphics emulation block 108, including:

- window x, window y registers 450, 452 specifying the coordinates of the display window (these coordinates may be copied from the memory [wx], memory [ly] values at the top of each frame),
- a window source y register 454 specifying the y coordinate for the source data for a window (this may start at zero at the top of the window),
- a background base pointer 456 that stores the base pointer for the current background RAM area (moves between background 1 and background 2, pages zero and one),
- a background pointer bank zero and background pointer bank one register 458, 460 specifying the base pointer for the current background RAM page zero and page one (these registers move between background 1 and background 2 areas),

  a window pointer bank zero register and a window pointer bank one register 462, 464 (these registers specify base pointers for the current

window RAM page zero and one respectively (they move between pages zero and one)),

- a character base pointer register 466 specifying the base pointer for the current character RAM area (moves between pages zero and one),
- a character RAM base pointer 467 specifying the base pointer for the current internal COLOR GAME BOY® RAM area 175 (0xC000-0xE000),
- a character bit mapped base pointer 468 specifying the base pointer for the "bit map-ized" character data 164 (this pointer moves between pages zero and one),
- a memory bank controller RAM base pointer 470 specifying the base pointer for the current cartridge RAM page 172,
- a character bit map index bank zero pointer and a character bit map index bank one pointer 472, 474 used as pointers to pre-sort an (addressing mode appropriate) array of pointers to the bank zero (bank one) "bit map-ized" character data.

[000129] Figure 24 shows an example illustration of a preferred embodiment emulated object attribute memory 173. In this example, the native platform includes an object attribute memory that maintains pointer and other information relating to characters to be displayed on the next frame. Preferred example embodiment emulator 100 includes an emulated object attribute memory 173 including an array of up to 40 objects each including y (vertical position), x (horizontal position), character (identifier) and attribute field. The bit-encoding of the attribute field information is also shown in Figure 24. An OAM base pointer 476 is used to function as a pre-allocated pointer to the emulated object attribute memory object 173.

[000130] Figure 25 shows an example video memory arrangement including an off screen memory buffer 168 and an on screen memory buffer 170. In the example

embodiment, the off screen memory buffer 168 is defined to be larger than the display size of the native platform. For example, the display size of a GAME BOY® or COLOR GAME BOY® is 160 pixels by 144 pixels high. In the example embodiment, off screen memory buffer 168 is defined to be 192 pixels wide by 160 pixels high -- leaving additional memory locations on all sides of the screen size buffer. A buffer zone of sixteen bytes or eight bytes is useful in improving efficiency. The use of screen memory buffers larger than screen size is an attempt to increase graphic drawing efficiency by eliminating clipping calculations and using the hardware BitBlt to transfer a subset of the memory buffer to displayed video memory. It may be desirable to use two off-screen buffers and one on-screen buffer in a classic "double-buffering" technique. Emulator 100 may draw to one buffer until it is complete, then switch to the other buffer. On the target hardware's vertical retrace interrupt, the emulator 100 copies the last-completed buffer to the on-screen area via BitBlt. This works well when a way is provided to implement the vertical retrace interrupt. As mentioned above, the "buffer zone" in the memory buffer is used to eliminate clipping calculations.

[000131] The example embodiment emulator 100 uses a hardware-assisted bit BLIT operation to copy the contents of the screen size buffer into on screen buffer 170. Such a bit-BLIT hardware-assisted operator can increase transfer times without corresponding increases in overhead. If a bit BLIT operation is not available, then a conventional direct memory access or other memory transfer can be used instead. In the example embodiment, a STPC_TARGET register 478 is used to specify whether a bit BLIT engine of the target platform is available and can be used (if one is not available, then a conventional memory copy function can be used instead).

[000132] Figure 26 shows some example additional graphics mode selectors used by emulator 100 in the preferred embodiment, including:

- a selector graphics engine register 480 that selects protected-mode memory for the graphics engine registers,

a selector off screen register 482 that selects protected mode memory for the off-screen video memory buffer 168 (two such selector registers can be used to indicate which one of two double buffered buffers is currently being drawn, with an index variable indicating this), and

- a selector on screen register 484 that selects protected mode memory for the on-screen video memory buffer 170.

[000133] Figure 27 shows an example screen layout of display 64 showing that the emulated display provided by on-screen buffer 170 may be smaller than the actual display area of display 64. In one example embodiment, the graphics adapter 62 and associated display 64 may provide a resolution of 320 pixels by 200 pixels, whereas emulator 100 produces an emulated image of 160 pixels by 144 pixels. Emulator 100 uses only a subset of display 64 to display emulated images in order to preserve aspect ratio.

[000134] Figure 28 shows an example set of graphics adapter 62 control constants that may be set to the VGA graphics adapter in order to set the graphics adapter's mode for use with emulator 100. Emulator 100 may be hard-coded to a particular graphics mode (320x200x16), an 8-bit color mode, or other mode available on the target hardware. If the 320x200x16 color mode (VESA mode 0x10E) works on a particular target platform, emulator 100 may use this mode exclusively -- and there will be no need for different control constants for the VGA.

[000135] Figure 29 shows example graphics engine register indices and associated example values. A screen pitch of 384 may also be defined as a constant in the example embodiment.

Example Color Palette Processing

[000136] In the example embodiment, the handling of color palettes can lead to efficiency problems. In the example native platforms, graphics characters are represented in a color lookup table (CLUT) format (i.e., the graphics characters

themselves include a reduced number of bits that are used to look up a color value in a color palette for display). See Figure 21. The example COLOR GAME BOY® native platform can display 56 colors on the screen nominally (eight palettes of four colors each for background, and eight palettes of four colors each for object characters minus transparency). It would seem therefore that with only 56 simultaneous colors on the screen at any one time, it would be possible to use a 320 x 200 x 8-bit VGA mode (13H) which would provide 256 different colors on the screen at once (much more than 56 colors). One possibility would be to simply add an offset into a VGA palette when using 8 bits of color. However, certain game developers for the COLOR GAME BOY® native platform change color palettes during horizontal blanking periods to achieve greater color variety on the screen. To emulate for these particular games, it is necessary to provide more than 256 colors on the screen at a time. A mode such as the VESA standard (320 x 216 bits) provides 65,000 different colors (RGB 565) -- about twice as much as the 32,000 colors the GAME BOY COLOR ® is capable of displaying. Thus, this video adapter mode is adequate for even applications that change their color palettes in the middle of a frame -- but using this mode doubles the amount of information that must be sent to the video memory per frame and costs processing time. Moreover, it also requires emulator 100 in the preferred embodiment to map color information (4 bits) into 32,000 colors. The 16-bit color resolution makes it desirable for emulator 100 to write 16-bit color palette information into the video adapter 62. This, in turn, necessitates a memory array of 16-bit numbers associated with the various color palettes. The color palettes (see Figure 4) can be accessed on a character-by-character basis, using pointers to apply color information to the "bit map-ized" character data 164 before the data is written to the display buffer 168.

[000137] While the invention has been described in connection with what is presently considered to be the most practical and preferred embodiment, it is to be understood that the invention is not to be limited to the disclosed embodiment, but on the contrary, is intended to cover various modifications and equivalent arrangements included within the scope of the appended claims.